NUMPY FOR BEGINNERS

AIOTLAB, September 2023

Table of Contents

Table of Contents

Introduction

- NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.
- Many computational packages providing scientific functionality use NumPy's array objects as one of the standard interface lingua francas for data exchange.
- Some of the things in Numpy:
  - ndarray, an efficient multidimensional array providing fast array-oriented arithmetic operations and flexible broadcasting capabilities.
  - Mathematical functions for fast operations on entire arrays of data without having to write loops.
  - Tools for reading/writing array data to disk and working with memory-mapped files.
  - Linear algebra, random number generation, and Fourier transform capabilities.
  - A C API for connecting NumPy with libraries written in C, C++, or FORTRAN

Introduction

- Having an understanding of NumPy arrays and array-oriented computing will help you use tools with array computing semantics, like pandas, much more effectively.
- One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data. There are a number of reasons for this:
  - NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects. NumPy's library of algorithms written in the C language can operate on this memory without any type checking or other overhead. NumPy arrays also use much less memory than built-in Python sequences.
  - NumPy operations perform complex computations on entire arrays without the need for Python **for** loops, which can be slow for large sequences. NumPy is faster than regular Python code because its C-based algorithms avoid overhead present with regular interpreted Python code.

Introduction

- The performance difference of Numpy and Python built-in **list**

```
my_arr = np.arange(1_000_000)
my_list = list(range(1_000_000))

%timeit my_arr2 = my_arr * 2
Output:
754 µs ± 12.4 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

%timeit my_list2 = [x * 2 for x in my_list]
Output:
37 ms ± 533 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Table of Contents

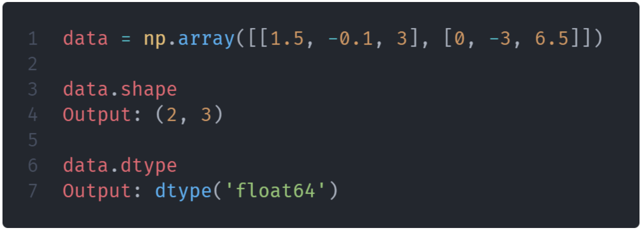
The NumPy ndarray: A Multidimensional Array Object

- One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python.
- Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

```
data = np.array([[1.5, -0.1, 3], [0, -3, 6.5]])

data * 10
Output: array([[ 15.,  -1.,  30.],
               [  0., -30.,  65.]])
```

The NumPy ndarray: A Multidimensional Array Object

- An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type.
- Every array has a **shape**, a tuple indicating the size of each dimension, and a **dtype**, an object describing the data type of the array:



```
data = np.array([[1.5, -0.1, 3], [0, -3, 6.5]])

data.shape
Output: (2, 3)

data.dtype
Output: dtype('float64')
```

The NumPy ndarray: A Multidimensional Array Object

- Creating ndarrays
  - The easiest way to create an array is to use the **array** function.
  - This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

```python
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)

arr1
Output: array([6. , 7.5, 8. , 0. , 1. ])
```

  - Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```python
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)

arr2
Output: array([[1, 2, 3, 4],
               [5, 6, 7, 8]])
```

The NumPy ndarray: A Multidimensional Array Object

- We can use **ndim** to get the dimension of the Numpy array.

```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)

arr2.ndim
Output: 2
```

- Unless explicitly specified, numpy.array tries to infer a good data type for the array that it creates.
- The data type is stored in a special **dtype** metadata object; for example.

```
arr1 = np.array([6, 7.5, 8, 0, 1])
arr2 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

arr1.dtype
Output: dtype('float64')

arr2.dtype
Output: dtype('int32')
```

The NumPy ndarray: A Multidimensional Array Object

- In addition to **numpy.array**, there are a number of other functions for creating new arrays.
- **numpy.zeros** and **numpy.ones** create arrays of 0s or 1s, respectively, with a given length or shape.
- **numpy.empty** creates an array without initializing its values to any particular value, it may contain nonzero "garbage" values.
- To create a higher dimensional array with these methods, pass a tuple for the shape.

```
np.zeros(10)
Output: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

np.zeros((3, 6))
Output: array([[0., 0., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0., 0.]])
```

The NumPy ndarray: A Multidimensional Array Object

```
np.ones(10)
Output: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

np.ones((3, 6))
Output: array([[1., 1., 1., 1., 1., 1.],
               [1., 1., 1., 1., 1., 1.],
               [1., 1., 1., 1., 1., 1.]])
```

```
np.empty((2, 3, 2))
Output: array([[[6.65058100e-312, 3.16202013e-322],
                [0.00000000e+000, 0.00000000e+000],
                [1.03977794e-312, 1.60050934e-047]],

               [[4.31241914e-033, 2.61997929e+180],
                [5.25335024e+170, 2.03839959e+184],
                [8.32414295e-071, 1.12100890e-047]]])
```

The NumPy ndarray: A Multidimensional Array Object

- **numpy.arange** is an array-valued version of the built-in Python **range** function

```
np.arange(15)

Output:
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

- **numpy.asarray** Convert input to ndarray, but do not copy if the input is already an ndarray

```
a = [1, 2]
type(np.asarray(a))

Output: numpy.ndarray
```

The NumPy ndarray: A Multidimensional Array Object

- **numpy.identity** create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

```
np.identity(3)

Output: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])
```

- **numpy.eye** create a square N x N matrix where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one.

```
np.eye(3)
Output: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])

np.eye(3, k=1)
Output: array([[0., 1., 0.],
               [0., 0., 1.],
               [0., 0., 0.]])
```

The NumPy ndarray: A Multidimensional Array Object

- Some important NumPy array creation functions

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a data type or explicitly specifying a data type; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list
`ones`, `ones_like`	Produce an array of all 1s with the given shape and data type; `ones_like` takes another array and produces a `ones` array of the same shape and data type
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty`, `empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full`, `full_like`	Produce an array of the given shape and data type with all values set to the indicated "fill value"; `full_like` takes another array and produces a filled array of the same shape and data type
`eye`, `identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

The NumPy ndarray: A Multidimensional Array Object

- Data Types for ndarrays
  - The data type or **dtype** is a special object containing the information (or metadata, data about data) the ndarray needs to interpret a chunk of memory as a particular type of data.

```
arr1 = np.array([1, 2, 3], dtype=np.float64)

arr1.dtype
Output: dtype('float64')
```

  - You can explicitly convert or cast an array from one data type to another using ndarray's **astype** method.

```
arr = np.array([1, 2, 3, 4, 5])
float_arr = arr.astype(np.float64)

float_arr.dtype
Output: dtype('float64')
```

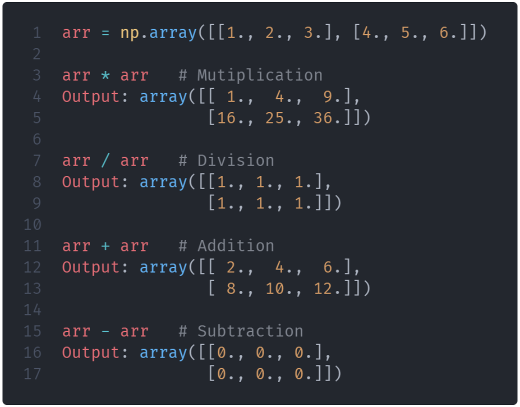
The NumPy ndarray: A Multidimensional Array Object

- When casting some floating-point numbers to be of integer data type, the decimal part wil be truncaed.

```
arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

arr.astype(np.int32)
Output: array([ 3, -1, -2,  0, 12, 10])
```

The NumPy ndarray: A Multidimensional Array Object

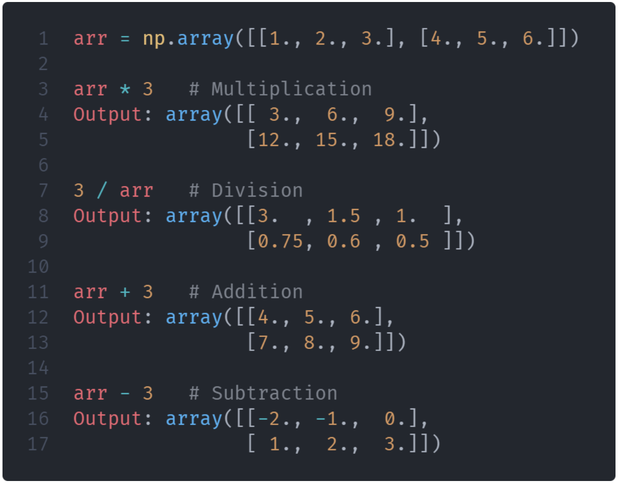
- Arithmetic with NumPy Arrays
  - Arrays are important because they enable you to express batch operations on data without writing any **for** loops. NumPy users call this vectorization.
  - Any arithmetic operations between equal-size arrays can apply the operation element-wise



```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])

arr * arr   # Mutiplication
Output: array([[ 1.,  4.,  9.],
               [16., 25., 36.]])

arr / arr   # Division
Output: array([[1., 1., 1.],
               [1., 1., 1.]])

arr + arr   # Addition
Output: array([[ 2.,  4.,  6.],
               [ 8., 10., 12.]])

arr - arr   # Subtraction
Output: array([[0., 0., 0.],
               [0., 0., 0.]])
```

The NumPy ndarray: A Multidimensional Array Object

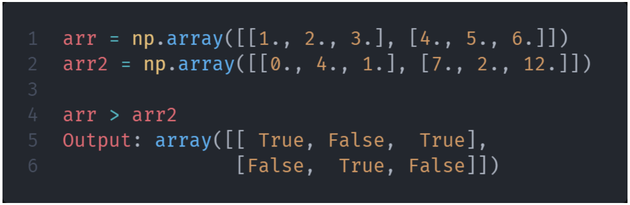
- Arithmetic operations with scalars propagate the scalar argument to each element in the array:



```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])

arr * 3   # Multiplication
Output: array([[ 3.,  6.,  9.],
               [12., 15., 18.]])

3 / arr   # Division
Output: array([[3.  , 1.5 , 1.  ],
               [0.75, 0.6 , 0.5 ]])

arr + 3   # Addition
Output: array([[4., 5., 6.],
               [7., 8., 9.]])

arr - 3   # Subtraction
Output: array([[-2., -1.,  0.],
               [ 1.,  2.,  3.]])
```

The NumPy ndarray: A Multidimensional Array Object

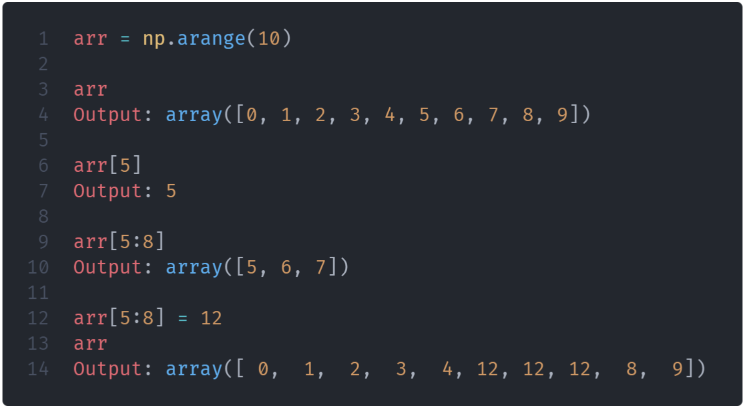
- Comparisons between arrays of the same size yield Boolean arrays:



```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

arr > arr2
Output: array([[ True, False,  True],
               [False,  True, False]])
```

The NumPy ndarray: A Multidimensional Array Object

- Basic Indexing and Slicing
  - NumPy array indexing is a deep topic, as there are many ways you may want to select a subset of your data or individual elements.
  - One-dimensional arrays are simple; on the surface they act similarly to Python lists:



```
arr = np.arange(10)

arr
Output: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

arr[5]
Output: 5

arr[5:8]
Output: array([5, 6, 7])

arr[5:8] = 12
arr
Output: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

The NumPy ndarray: A Multidimensional Array Object

- An important first distinction from Python's built-in lists is that array slices are **views** on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

```
arr = np.arange(10)

arr
Output: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

arr_slice = arr[5:8]
arr_slice
Output: array([5, 6, 7])

arr_slice[:] = 100
arr
Output: array([  0,   1,   2,   3,   4, 100, 100, 100,   8,   9])
```

The NumPy ndarray: A Multidimensional Array Object

- With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

arr2d[1]
Output: array([4, 5, 6])
```

- Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
print(arr2d[0][2])
Output: 3

print(arr2d[0, 2])
Output: 3
```

The NumPy ndarray: A Multidimensional Array Object

- In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the $2 \times 2 \times 3$ array:

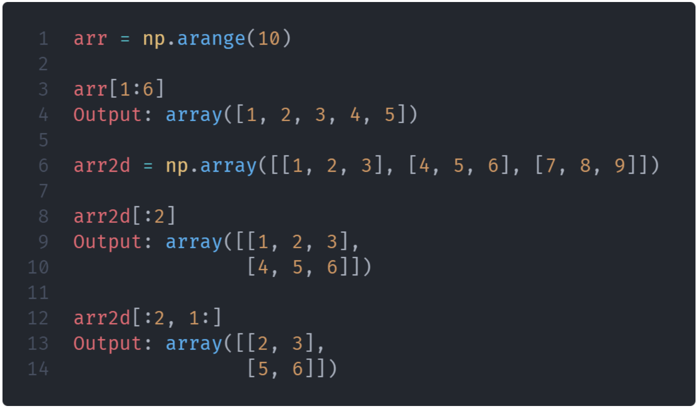
```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

arr3d[0]
Output: array([[1, 2, 3],
               [4, 5, 6]])
```

- Both scalar values and arrays can be assigned to **arr3d[0]**:

```
arr3d[0] = 42

arr3d
Output: array([[[42, 42, 42],
                [42, 42, 42]],

               [[ 7,  8,  9],
                [10, 11, 12]]])
```

The NumPy ndarray: A Multidimensional Array Object

- Indexing with slices
  - Like one-dimensional objects such as Python lists, ndarrays can be sliced with the familiar syntax:



```python
arr = np.arange(10)

arr[1:6]
Output: array([1, 2, 3, 4, 5])

arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

arr2d[:2]
Output: array([[1, 2, 3],
               [4, 5, 6]])

arr2d[:2, 1:]
Output: array([[2, 3],
               [5, 6]])
```

The NumPy ndarray: A Multidimensional Array Object

- Boolean Indexing
  - Let's consider an example where we have some data in an array and an array of names with duplicates:

```python
names = np.array(["Bob", "Joe", "Will", "Bob",
                  "Will", "Joe", "Joe"])

data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0],
                 [1, 2], [-12, -4], [3, 4]])
```

  - Suppose each name corresponds to a row in the data array and we wanted to select all the rows with the corresponding name **"Bob"**. Like arithmetic operations, comparisons (such as **==**) with arrays are also vectorized. Thus, comparing **names** with the string **"Bob"** yields a Boolean array:

```python
names == "Bob"

Output: array([ True, False, False,  True, False, False, False])
```

The NumPy ndarray: A Multidimensional Array Object

- This Boolean array can be passed when indexing the array: Boolean array:

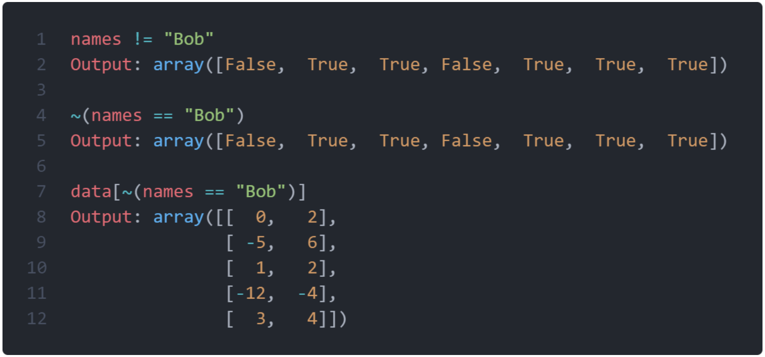
```
data[names == "Bob"]

Output: array([[4, 7],
               [0, 0]])
```

- The Boolean array must be of the same length as the array axis it's indexing. You can even mix and match Boolean arrays with slices or integers:

```
data[names == "Bob", 1:]
Output: array([[7],
               [0]])

data[names == "Bob", 1]
Output: array([7, 0])
```

The NumPy ndarray: A Multidimensional Array Object

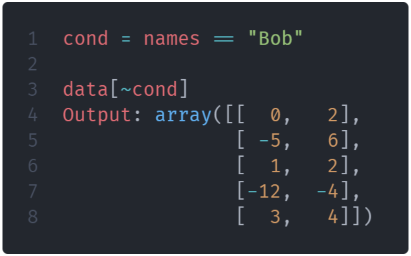
- To select everything but **"Bob"** you can either use **!=** or negate the condition using ~:



```
names != "Bob"
Output: array([False,  True,  True, False,  True,  True,  True])

~(names == "Bob")
Output: array([False,  True,  True, False,  True,  True,  True])

data[~(names == "Bob")]
Output: array([[  0,   2],
               [ -5,   6],
               [  1,   2],
               [-12,  -4],
               [  3,   4]])
```

The NumPy ndarray: A Multidimensional Array Object

- The ~ operator can be useful when you want to invert a Boolean array referenced by a variable



```
cond = names == "Bob"

data[~cond]
Output: array([[  0,   2],
               [ -5,   6],
               [  1,   2],
               [-12,  -4],
               [  3,   4]])
```

The NumPy ndarray: A Multidimensional Array Object

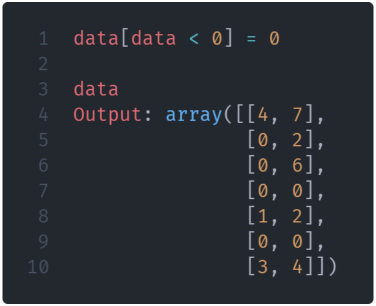
- To select two of the three names to combine multiple Boolean conditions, use Boolean arithmetic operators like **&** (and) and | (or):

```
mask = (names == "Bob") | (names == "Will")

mask
Output: array([ True, False,  True,  True,  True, False, False])

data[mask]
Output: array([[ 4,  7],
               [-5,  6],
               [ 0,  0],
               [ 1,  2]])
```

- Note: The Python keywords **and** and **or** do not work with Boolean arrays. Use **&** (and) and | (or) instead.

The NumPy ndarray: A Multidimensional Array Object

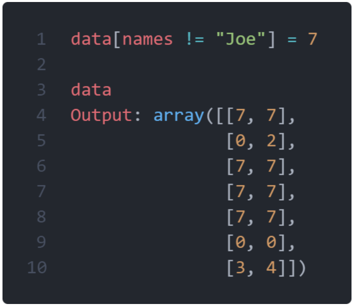
- Setting values with Boolean arrays works by substituting the value or values on the righthand side into the locations where the Boolean array's values are **True**. For example, to set all of the negative values in data to 0, we need only do:



```
data[data < 0] = 0

data
Output: array([[4, 7],
               [0, 2],
               [0, 6],
               [0, 0],
               [1, 2],
               [0, 0],
               [3, 4]])
```

The NumPy ndarray: A Multidimensional Array Object

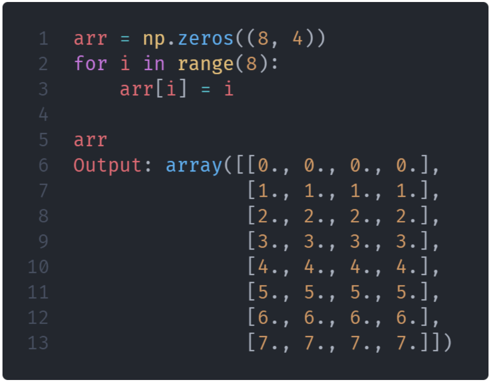
- You can also set whole rows or columns using a one-dimensional Boolean array:



```
data[names != "Joe"] = 7

data
Output: array([[7, 7],
               [0, 2],
               [7, 7],
               [7, 7],
               [7, 7],
               [0, 0],
               [3, 4]])
```

The NumPy ndarray: A Multidimensional Array Object

- Fancy Indexing
  - Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had an 8×4 array:



```
arr = np.zeros((8, 4))
for i in range(8):
    arr[i] = i

arr
Output: array([[0., 0., 0., 0.],
               [1., 1., 1., 1.],
               [2., 2., 2., 2.],
               [3., 3., 3., 3.],
               [4., 4., 4., 4.],
               [5., 5., 5., 5.],
               [6., 6., 6., 6.],
               [7., 7., 7., 7.]])
```

The NumPy ndarray: A Multidimensional Array Object

- To select a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

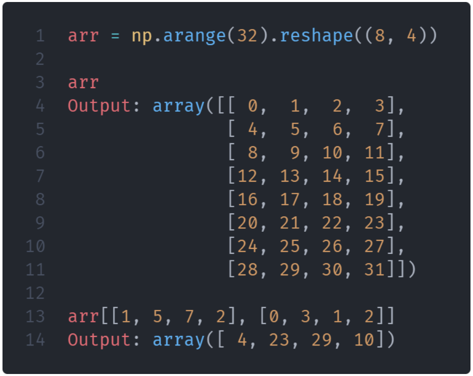
```
arr[[4, 3, 0, 6]]

Output: array([[4., 4., 4., 4.],
               [3., 3., 3., 3.],
               [0., 0., 0., 0.],
               [6., 6., 6., 6.]])
```

- Using negative indices selects rows from the end:

```
arr[[-3, -5, -7]]

Output: array([[5., 5., 5., 5.],
               [3., 3., 3., 3.],
               [1., 1., 1., 1.]])
```

The NumPy ndarray: A Multidimensional Array Object

- Passing multiple index arrays does something slightly different; it selects a one-dimensional array of elements corresponding to each tuple of indices:



```
arr = np.arange(32).reshape((8, 4))

arr
Output: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11],
               [12, 13, 14, 15],
               [16, 17, 18, 19],
               [20, 21, 22, 23],
               [24, 25, 26, 27],
               [28, 29, 30, 31]])

arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Output: array([ 4, 23, 29, 10])
```

- Here the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected. The result of fancy indexing with as many integer arrays as there are axes is always one-dimensional.

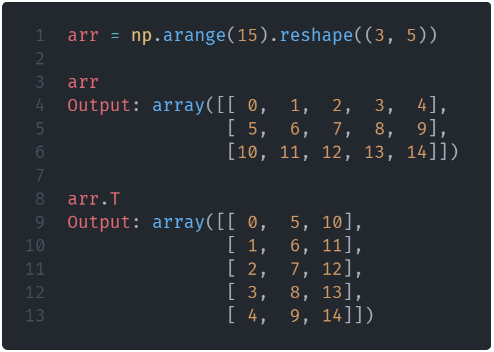
The NumPy ndarray: A Multidimensional Array Object

- The behavior of fancy indexing in the below case is a bit different, which is the rectangular region formed by selecting a subset of the matrix's rows and columns:

```
arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]

Output: array([[ 4,  7,  5,  6],
               [20, 23, 21, 22],
               [28, 31, 29, 30],
               [ 8, 11,  9, 10]])
```

The NumPy ndarray: A Multidimensional Array Object

- Transposing Arrays and Swapping Axes
  - Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything. Arrays have the **transpose** method and the special **T** attribute:



```
arr = np.arange(15).reshape((3, 5))

arr
Output: array([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14]])

arr.T
Output: array([[ 0,  5, 10],
               [ 1,  6, 11],
               [ 2,  7, 12],
               [ 3,  8, 13],
               [ 4,  9, 14]])
```

The NumPy ndarray: A Multidimensional Array Object

- When doing matrix computations, you may do this very often—for example, when computing the inner matrix product using **numpy.dot**:

```
arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2], [-1, 0, -1], [1, 0, 1]])

np.dot(arr, arr)
Output:
ValueError: shapes (5,3) and (5,3) not aligned: 3 (dim 1) ≠ 5 (dim 0)

np.dot(arr.T, arr)
Output: array([[39, 20, 12],
               [20, 14,  2],
               [12,  2, 10]])
```

The NumPy ndarray: A Multidimensional Array Object

- The **@** infix operator is another way to do matrix multiplication

```
arr.T @ arr

Output: array([[39, 20, 12],
               [20, 14,  2],
               [12,  2, 10]])
```

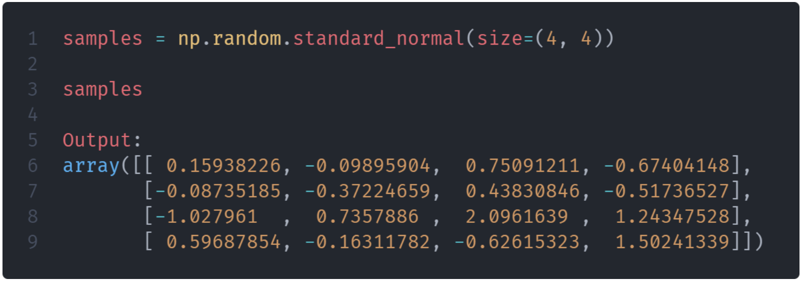
- Simple transposing with **.T** is a special case of swapping axes. ndarray has the method **swapaxes**, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

```
arr.swapaxes(0, 1)

Output: array([[ 0,  1,  6, -1,  1],
               [ 1,  2,  3,  0,  0],
               [ 0, -2,  2, -1,  1]])
```

Table of Contents

Pseudorandom Number Generation

- The **numpy.random** module supplements the built-in Python **random** module with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions
- Can get a 4 × 4 array of samples from the standard normal distribution using **numpy.random.standard_normal**



```
samples = np.random.standard_normal(size=(4, 4))

samples

Output:
array([[ 0.15938226, -0.09895904,  0.75091211, -0.67404148],
       [-0.08735185, -0.37224659,  0.43830846, -0.51736527],
       [-1.027961  ,  0.7357886 ,  2.0961639 ,  1.24347528],
       [ 0.59687854, -0.16311782, -0.62615323,  1.50241339]])
```

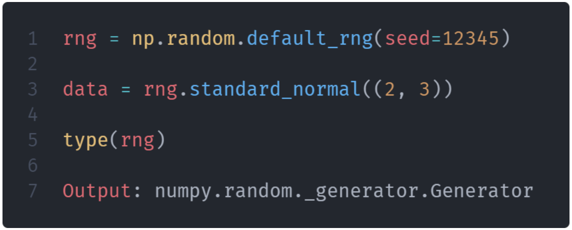
Pseudorandom Number Generation

- Python's built-in **random** module, by contrast, samples only one value at a time
- **numpy.random** is well over an order of magnitude faster for generating very large samples

```
from random import normalvariate

N = 1_000_000

%timeit samples = [normalvariate(0, 1) for _ in range(N)]

%timeit np.random.standard_normal(N)

Output:
477 ms ± 3.01 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
16.4 ms ± 88 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Pseudorandom Number Generation

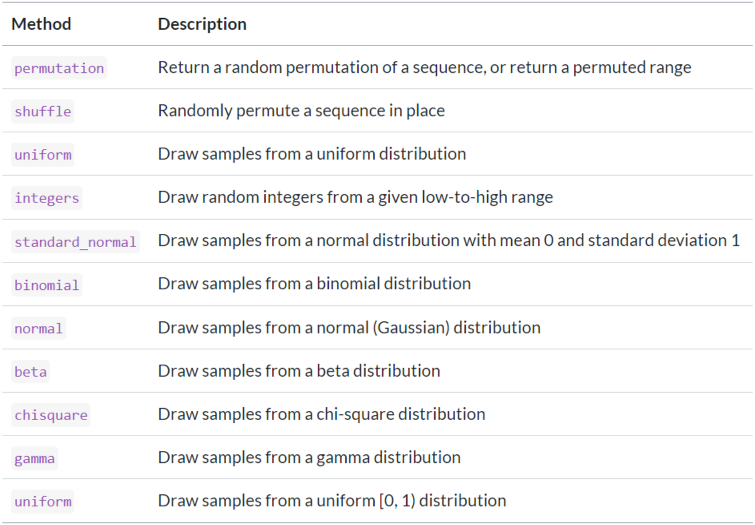
- These random numbers are not truly random (rather, **pseudorandom**) but instead are generated by a configurable random number generator that determines deterministically what values are created
    - The **seed** argument is what determines the initial state of the generator, and the state changes each time the **rng** object is used to generate data
    - The generator object **rng** is also isolated from other code which might use the **numpy.random** module



```
rng = np.random.default_rng(seed=12345)

data = rng.standard_normal((2, 3))

type(rng)

Output: numpy.random._generator.Generator
```

Pseudorandom Number Generation

- Partial list of methods available on random generator objects like **rng**



Method	Description
`permutation`	Return a random permutation of a sequence, or return a permuted range
`shuffle`	Randomly permute a sequence in place
`uniform`	Draw samples from a uniform distribution
`integers`	Draw random integers from a given low-to-high range
`standard_normal`	Draw samples from a normal distribution with mean 0 and standard deviation 1
`binomial`	Draw samples from a binomial distribution
`normal`	Draw samples from a normal (Gaussian) distribution
`beta`	Draw samples from a beta distribution
`chisquare`	Draw samples from a chi-square distribution
`gamma`	Draw samples from a gamma distribution
`uniform`	Draw samples from a uniform [0, 1) distribution

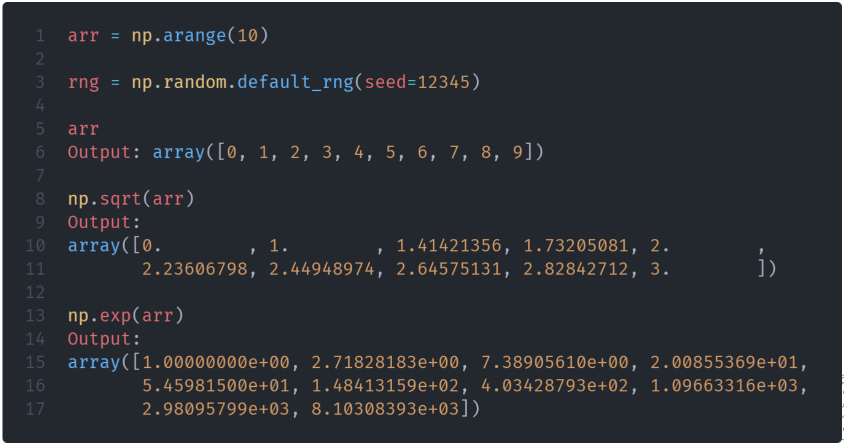
Table of Contents

Universal Functions - Fast Element-Wise Array Functions

- A universal function, or **ufunc**, is a function that performs element-wise operations on data in ndarrays
- Can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results

Universal Functions - Fast Element-Wise Array Functions

- Many *ufuncs* are simple element-wise transformations, like **numpy.sqrt** or **numpy.exp**
- These are referred to as **unary** *ufuncs*



```
arr = np.arange(10)

rng = np.random.default_rng(seed=12345)

arr
Output: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

np.sqrt(arr)
Output:
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])

np.exp(arr)
Output:
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])
```

Universal Functions - Fast Element-Wise Array Functions

- Others, such as **numpy.add** or **numpy.maximum**, take two arrays (thus, **binary** *ufuncs*) and return a single array as the result

```
x = rng.standard_normal(8)
y = rng.standard_normal(8)

x
Output:
array([-1.42382504,  1.26372846, -0.87066174, -0.25917323, -0.07534331,
       -0.74088465, -1.3677927 ,  0.6488928 ])

y
Output:
array([ 0.36105811, -1.95286306,  2.34740965,  0.96849691, -0.75938718,
        0.90219827, -0.46695317, -0.06068952])

np.add(x, y)
Output:
array([-1.06276692, -0.6891346 ,  1.47674792,  0.70932367, -0.83473049,
        0.16131362, -1.83474588,  0.58820328])

np.maximum(x, y)
Output:
array([ 0.36105811,  1.26372846,  2.34740965,  0.96849691, -0.07534331,
        0.90219827, -0.46695317,  0.6488928 ])
```

Universal Functions - Fast Element-Wise Array Functions

- While not common, a *ufunc* can return **multiple arrays**
  - **numpy.modf**: a vectorized version of the built-in Python **math.modf**
  - Returns the fractional and integral parts of a floating-point array

```
arr = rng.standard_normal(7) * 5

arr
Output:
array([ 3.94422172, -6.28334067,  2.87928757,  6.99489497,  6.6114903 ,
       -1.49849258,  4.51459671])

remainder, whole_part = np.modf(arr)

remainder
Output:
array([ 0.94422172, -0.28334067,  0.87928757,  0.99489497,  0.6114903 ,
       -0.49849258,  0.51459671])

whole_part
Output:
array([ 3., -6.,  2.,  6.,  6., -1.,  4.])
```

Universal Functions - Fast Element-Wise Array Functions

- *Ufuncs* can accept an optional **out** argument that allows them to assign their results into an existing array rather than create a new one

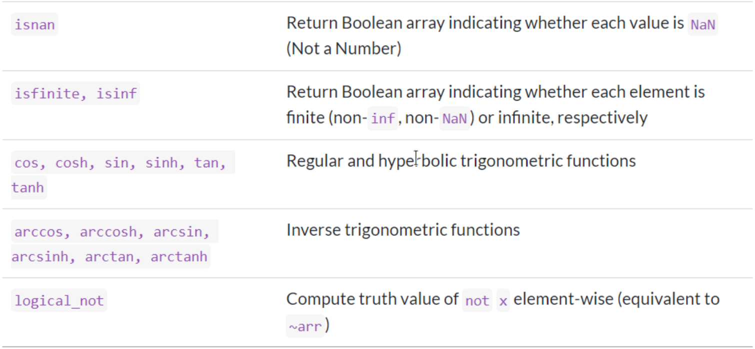
```
arr
Output:
array([ 3.94422172, -6.28334067,  2.87928757,  6.99489497,  6.6114903 ,
       -1.49849258,  4.51459671])

out = np.zeros_like(arr)

np.add(arr, 1)
Output:
array([ 4.94422172, -5.28334067,  3.87928757,  7.99489497,  7.6114903 ,
       -0.49849258,  5.51459671])

np.add(arr, 1, out=out)
Output:
array([ 4.94422172, -5.28334067,  3.87928757,  7.99489497,  7.6114903 ,
       -0.49849258,  5.51459671])

out
Output:
array([ 4.94422172, -5.28334067,  3.87928757,  7.99489497,  7.6114903 ,
       -0.49849258,  5.51459671])
```

Universal Functions - Fast Element-Wise Array Functions

- Some **unary** *ufuncs*

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as separate arrays

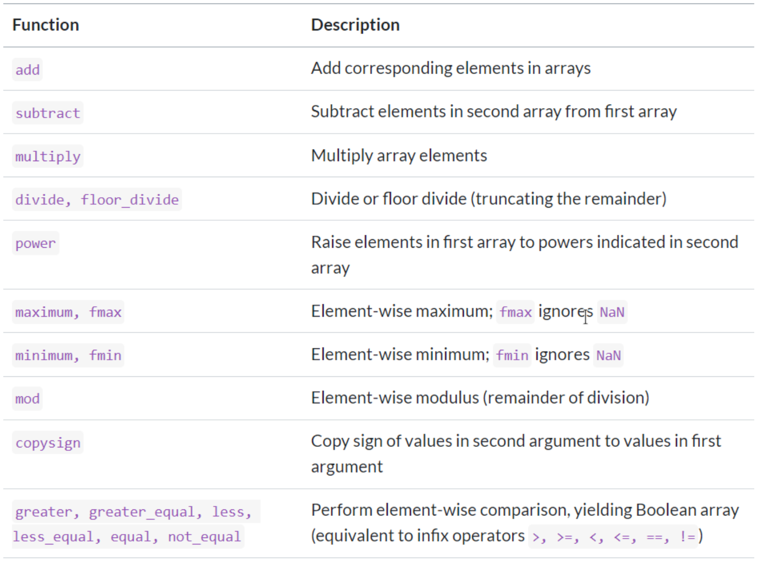
Universal Functions - Fast Element-Wise Array Functions



`isnan`	Return Boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return Boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not` `x` element-wise (equivalent to `~arr`)

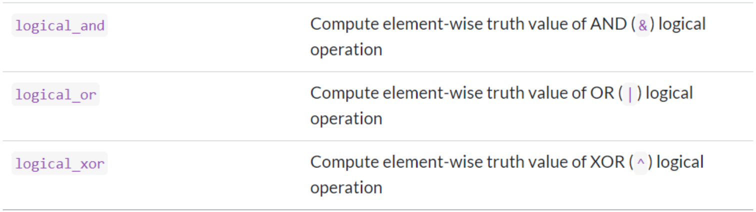
Universal Functions - Fast Element-Wise Array Functions

- Some **binary** *ufuncs*



Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum; `fmax` ignores `NaN`
`minimum, fmin`	Element-wise minimum; `fmin` ignores `NaN`
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding Boolean array (equivalent to infix operators `>, >=, <, <=, ==, !=`)

Universal Functions - Fast Element-Wise Array Functions



`logical_and`	Compute element-wise truth value of AND (`&`) logical operation
`logical_or`	Compute element-wise truth value of OR (`|`) logical operation
`logical_xor`	Compute element-wise truth value of XOR (`^`) logical operation

Table of Contents

Array-Oriented Programming with Arrays

- Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops
  - This practice of **replacing explicit loops** with array expressions is referred to by some people as **Vectorization**
  - In general, vectorized array operations will usually be **significantly faster** than their pure Python equivalents
  - Biggest impact in any kind of numerical computations

Array-Oriented Programming with Arrays

- Suppose we wished to evaluate the function $\sqrt{x^2 + y^2}$ across a regular grid of values
- The **numpy.meshgrid** function takes two one-dimensional arrays and produces two two-dimensional matrices corresponding to all pairs of **(x, y)** in the two arrays

Array-Oriented Programming with Arrays

```
points = np.arange(-5, 5, 0.01)
xs, ys = np.meshgrid(points, points)

xs
Output:
array([[-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       ...,
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99]])

ys
Output:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       ...,
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])

z = np.sqrt(xs ** 2 + ys ** 2)

z
Output:
array([[7.07106781, 7.06400028, 7.05693985, ..., 7.04988652, 7.05693985,
        7.06400028],
       ...,
       [7.06400028, 7.05692568, 7.04985815, ..., 7.04279774, 7.04985815,
        7.05692568]])
```

Expressing Conditional Logic as Array Operations

- The **numpy.where** function is a vectorized version of the ternary expression **x if condition else y**
- Suppose we had a Boolean array and 2 arrays of values. Suppose we wanted to take a value from **xarr** whenever the corresponding value in **cond** is **True**, and otherwise take the value from **yarr**
  - Using list comprehension has multiple problems
    - Slow for large arrays (because all the work is being done in interpreted Python code)
    - Will not work with multidimensional arrays
  - NumPy's **where** can perform said action with a single function call

Expressing Conditional Logic as Array Operations

```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])

result = [(x if c else y)
            for x, y, c in zip(xarr, yarr, cond)]

result
Output: [1.1, 2.2, 1.3, 1.4, 2.5]
```

```
result = np.where(cond, xarr, yarr)

result
Output: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Expressing Conditional Logic as Array Operations

- The **2nd** and **3rd** arguments to **numpy.where** don't need to be arrays; one or both of them can be *scalars*
  - A typical use of **where** in data analysis is to produce a new array of values based on another array
  - Suppose we had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with –2. This is possible to do with **numpy.where**

Expressing Conditional Logic as Array Operations

```
arr = rng.standard_normal((4, 4))

arr
Output:
array([[-1.62158273, -0.15818926,  0.44948393, -1.34360107],
       [-0.08168759,  1.72473993,  2.61815943,  0.77736134],
       [ 0.8286332 , -0.95898831, -1.20938829, -1.41229201],
       [ 0.54154683,  0.7519394 , -0.65876032, -1.22867499]])

arr > 0
Output:
array([[False, False,  True, False],
       [False,  True,  True,  True],
       [ True, False, False, False],
       [ True,  True, False, False]])

np.where(arr > 0, 2, -2)
Output:
array([[-2, -2,  2, -2],
       [-2,  2,  2,  2],
       [ 2, -2, -2, -2],
       [ 2,  2, -2, -2]])
```

Expressing Conditional Logic as Array Operations

- Can combine scalars and arrays when using **numpy.where**

```
np.where(arr > 0, 2, arr) # set only positive values to 2
Output:
array([[-1.62158273, -0.15818926,  2.        , -1.34360107],
       [-0.08168759,  2.        ,  2.        ,  2.        ],
       [ 2.        , -0.95898831, -1.20938829, -1.41229201],
       [ 2.        ,  2.        , -0.65876032, -1.22867499]])
```

Mathematical and Statistical Methods

- Can use aggregations (sometimes called *reductions*) like **sum**, **mean**, and **std** (standard deviation) either by calling the array instance method or using the top-level NumPy function
- When using the NumPy function, like **numpy.sum**, we have to pass the array we want to aggregate as the **first argument**

Mathematical and Statistical Methods

```
arr = rng.standard_normal((5, 4))

arr
Output:
array([[ 0.25755777,  0.31290292, -0.13081169,  1.26998312],
       [-0.09296246, -0.06615089, -1.10821447,  0.13595685],
       [ 1.34707776,  0.06114402,  0.0709146 ,  0.43365454],
       [ 0.27748366,  0.53025239,  0.53672097,  0.61835001],
       [-0.79501746,  0.30003095, -1.60270159,  0.26679883]])

arr.mean()
Output: 0.13114849172877924

np.mean(arr)
Output: 0.13114849172877924

arr.sum()
Output: 2.622969834575585

arr.std()
Output: 0.674973871220971
```

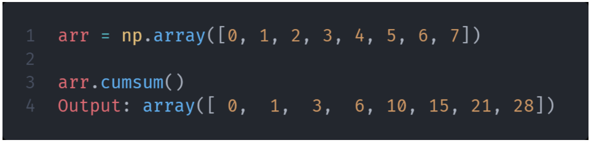
Mathematical and Statistical Methods

- Functions like **mean** and **sum** take an optional **axis** argument that computes the statistic over the given axis, resulting in an array with one less dimension
  - **axis=0**: Across the rows
  - **axis=1**: Across the columns

```
arr.mean(axis=0)
Output: array([ 0.19882786,  0.22763588, -0.44681844,  0.54494867])

arr.mean(axis=1)
Output: array([ 0.42740803, -0.28284274,  0.47819773,  0.49070176, -0.45772232])

arr.sum(axis=0)
Output: array([ 0.99413928,  1.13817938, -2.23409218,  2.72474335])

arr.sum(axis=1)
Output: array([ 1.70963212, -1.13137096,  1.91279092,  1.96280703, -1.83088927])
```

Mathematical and Statistical Methods

- Other methods like **cumsum** and **cumprod** do not aggregate, instead producing an array of the intermediate results



```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

arr.cumsum()
Output: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

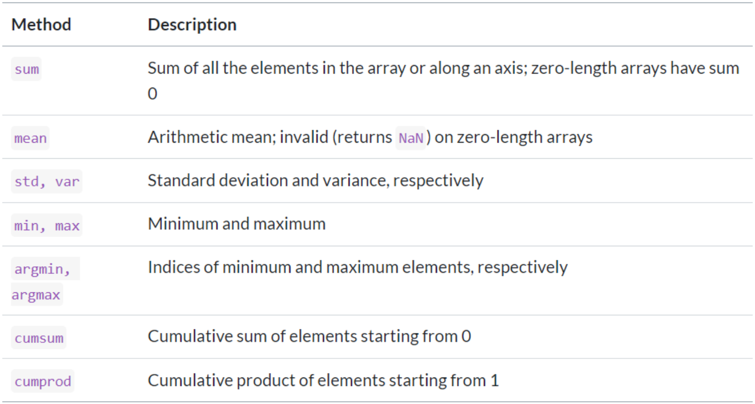
Mathematical and Statistical Methods

- In multidimensional arrays, accumulation functions like **cumsum** return an array of the same size but with the partial aggregates computed along the indicated axis according to each lower dimensional slice
    - **arr.cumsum(axis=0)**: The cumulative sums along the rows
    - **arr.cumsum(axis=1)**: The cumulative sums along the columns

```
arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

arr
Output: array([[0, 1, 2],
               [3, 4, 5],
               [6, 7, 8]])

arr.cumsum(axis=0)
Output: array([[ 0,  1,  2],
               [ 3,  5,  7],
               [ 9, 12, 15]])

arr.cumsum(axis=1)
Output: array([[ 0,  1,  3],
               [ 3,  7, 12],
               [ 6, 13, 21]])
```

Mathematical and Statistical Methods

- Basic array statistical methods



Method	Description
`sum`	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
`mean`	Arithmetic mean; invalid (returns `NaN`) on zero-length arrays
`std, var`	Standard deviation and variance, respectively
`min, max`	Minimum and maximum
`argmin, argmax`	Indices of minimum and maximum elements, respectively
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

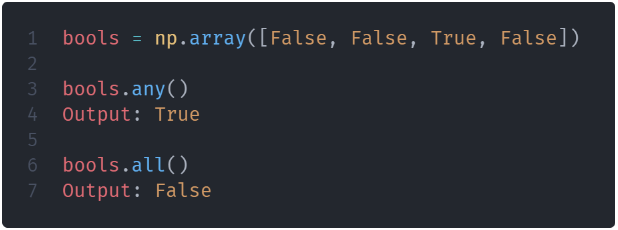
Methods for Boolean Arrays

- Boolean values are coerced to 1 (*True*) and 0 (*False*) in the preceding methods
    - Thus, **sum** is often used as a means of counting *True* values in a Boolean array
    - The parentheses here in the expression $(arr > 0).sum()$ are necessary to be able to call **sum()** on the temporary result of $arr > 0$

```
arr = rng.standard_normal(100)

(arr > 0).sum() # Number of positive values
Output: 49

(arr <= 0).sum() # Number of non-positive values
Output: 51
```

Methods for Boolean Arrays

- Two additional methods, **any** and **all**, are useful especially for Boolean arrays
  - **any** tests whether **one or more** values in an array is *True*
  - **all** checks if **every** value is *True*
  - These methods also work with non-Boolean arrays, where nonzero elements are treated as *True*



```python
bools = np.array([False, False, True, False])

bools.any()
Output: True

bools.all()
Output: False
```

Sorting

- NumPy arrays can be sorted in place with the **sort** method

```
arr = rng.standard_normal(6)

arr
Output: array([-1.0198852 ,  0.01875261, -1.89426422,
               -0.75500166,  0.75619774, -1.04246201])

arr.sort()

arr
Output: array([-1.89426422, -1.04246201, -1.0198852 ,
               -0.75500166,  0.01875261,  0.75619774])
```

Sorting

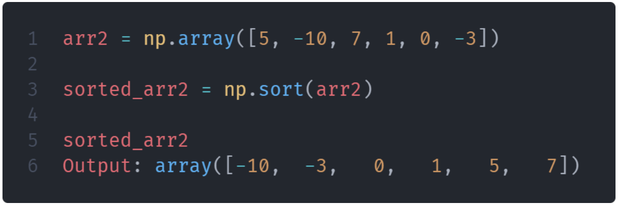
- Can sort each one-dimensional section of values in a multidimensional array in place along an axis by passing the axis number to **sort**
    - **arr.sort(axis=0)** sorts the values within each column
    - **arr.sort(axis=1)** sorts across each row

Sorting

```python
arr = rng.standard_normal((5, 3))
arr
Output: array([[-0.03425814, -0.3551683 , -0.37842837],
               [ 0.19064869,  0.48439629,  1.23026775],
               [ 0.83297062, -0.56494175,  1.41469601],
               [ 1.24828122, -1.5589481 ,  0.66523259],
               [ 0.82559517,  0.96631883,  0.5471753 ]])

arr.sort(axis=0)
arr
Output: array([[-0.03425814, -1.5589481 , -0.37842837],
               [ 0.19064869, -0.56494175,  0.5471753 ],
               [ 0.82559517, -0.3551683 ,  0.66523259],
               [ 0.83297062,  0.48439629,  1.23026775],
               [ 1.24828122,  0.96631883,  1.41469601]])

arr.sort(axis=1)
arr
Output: array([[-1.5589481 , -0.37842837, -0.03425814],
               [-0.56494175,  0.19064869,  0.5471753 ],
               [-0.3551683 ,  0.66523259,  0.82559517],
               [ 0.48439629,  0.83297062,  1.23026775],
               [ 0.96631883,  1.24828122,  1.41469601]])
```

Sorting

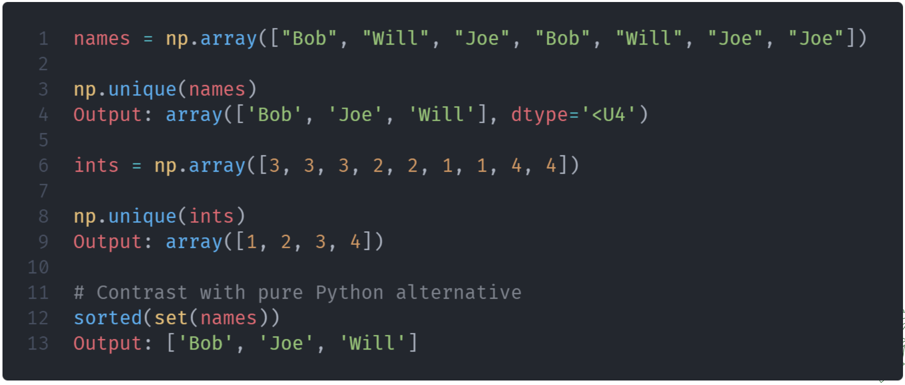
- The top-level method **numpy.sort** returns a sorted *copy* of an array (like the Python built-in function **sorted**) instead of modifying the array in place



```
arr2 = np.array([5, -10, 7, 1, 0, -3])

sorted_arr2 = np.sort(arr2)

sorted_arr2
Output: array([-10,  -3,   0,   1,   5,   7])
```

Unique and Other Set Logic

- NumPy has some basic set operations for one-dimensional ndarrays
- A commonly used one is **numpy.unique**, which returns the sorted unique values in an array
- In many cases, the NumPy version is faster and returns a NumPy array rather than a Python list



```
names = np.array(["Bob", "Will", "Joe", "Bob", "Will", "Joe", "Joe"])

np.unique(names)
Output: array(['Bob', 'Joe', 'Will'], dtype='<U4')

ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

np.unique(ints)
Output: array([1, 2, 3, 4])

# Contrast with pure Python alternative
sorted(set(names))
Output: ['Bob', 'Joe', 'Will']
```

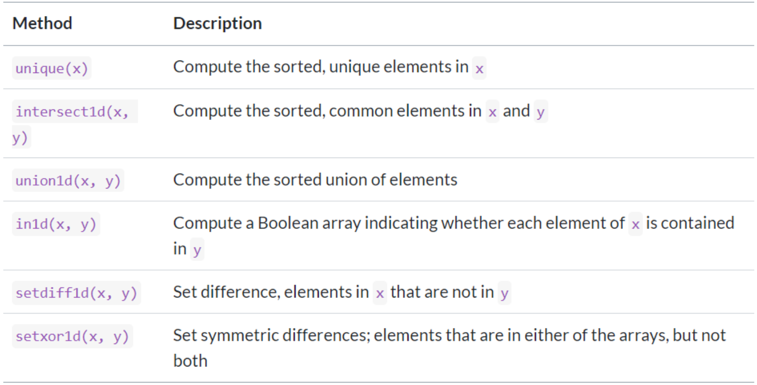
Unique and Other Set Logic

- **numpy.in1d**, tests *membership* of the values in one array in another, returning a Boolean array

```
values = np.array([6, 0, 0, 3, 2, 5, 6])

np.in1d(values, [2, 3, 6])
Output: array([ True, False, False,  True,  True, False,  True])
```

Unique and Other Set Logic

- Some array set operations

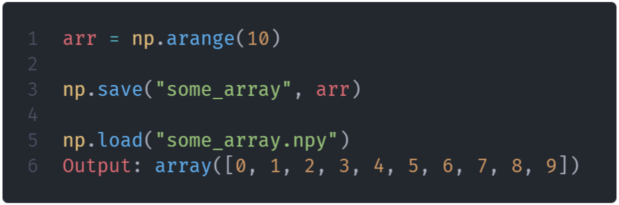


Method	Description
`unique(x)`	Compute the sorted, unique elements in `x`
`intersect1d(x, y)`	Compute the sorted, common elements in `x` and `y`
`union1d(x, y)`	Compute the sorted union of elements
`in1d(x, y)`	Compute a Boolean array indicating whether each element of `x` is contained in `y`
`setdiff1d(x, y)`	Set difference, elements in `x` that are not in `y`
`setxor1d(x, y)`	Set symmetric differences; elements that are in either of the arrays, but not both

Table of Contents

File Input-Output with Arrays

- NumPy is able to save and load data to and from disk in some text or binary formats
- **numpy.save** and **numpy.load** are the two workhorse functions for efficiently saving and loading array data on disk
  - Arrays are saved by default in an uncompressed raw binary format with file extension *.npy*
  - If the file path does not already end in *.npy*, the extension will be appended



```
arr = np.arange(10)

np.save("some_array", arr)

np.load("some_array.npy")
Output: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

File Input-Output with Arrays

- Can save multiple arrays in an uncompressed archive using **numpy.savez** and passing the arrays as keyword arguments

```
np.savez("array_archive.npz", a=arr, b=arr)
```

- When loading an *.npz* file, we get back a dictionary-like object that loads the individual arrays lazily

```
arch = np.load("array_archive.npz")

arch
Output: NpzFile 'array_archive.npz' with keys: a, b

arch["b"]
Output: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

File Input-Output with Arrays

- If our data compresses well, we may wish to use **numpy.savez_compressed** instead

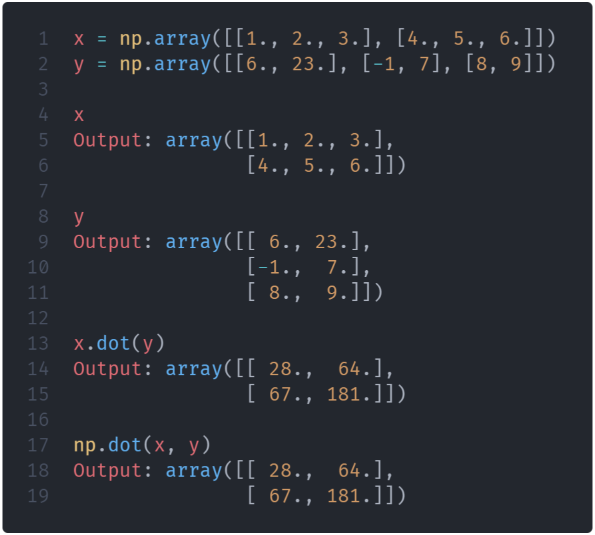
```python
np.savez_compressed("arrays_compressed.npz", a=arr, b=arr)
```

Table of Contents

Linear Algebra

- Linear algebra operations, like matrix multiplication, decomposition, determinants, and other square matrix math, are an important part of many array libraries
  - Multiplying two two-dimensional arrays with is an element-wise product
  - Matrix multiplications require either using the **dot** function or the **@** infix operator
  - **dot** is both an array method and a function in the **numpy** namespace for doing matrix multiplication
  - **x.dot(y)** is equivalent to **np.dot(x, y)**

Linear Algebra



```
x = np.array([[1., 2., 3.], [4., 5., 6.]])
y = np.array([[6., 23.], [-1, 7], [8, 9]])

x
Output: array([[1., 2., 3.],
               [4., 5., 6.]])

y
Output: array([[ 6., 23.],
               [-1.,  7.],
               [ 8.,  9.]])

x.dot(y)
Output: array([[ 28.,  64.],
               [ 67., 181.]])

np.dot(x, y)
Output: array([[ 28.,  64.],
               [ 67., 181.]])
```

Linear Algebra

- A matrix product between a two-dimensional array and a suitably sized one-dimensional array results in a one-dimensional array

```
x @ np.ones(3)
Output: array([ 6., 15.])
```

Linear Algebra

- **numpy.linalg** has a standard set of matrix decompositions and things like inverse and determinant
- The expression **X.T.dot(X)** computes the dot product of **X** with its transpose **X.T**

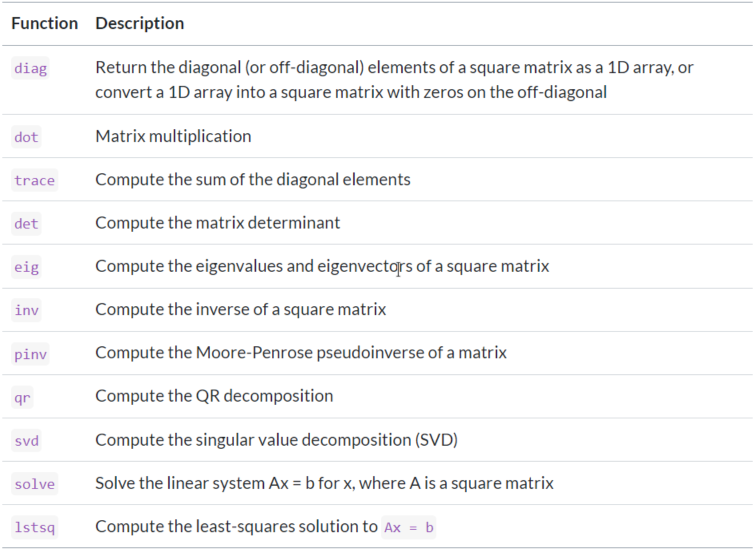
```python
from numpy.linalg import inv, qr

X = rng.standard_normal((5, 5))

mat = X.T @ X

inv(mat)
Output:
array([[ 0.66262641, -0.7241881 , -0.20371883, -0.22638348,  0.07681744],
       [-0.7241881 ,  1.81660322,  0.09637542,  0.66394756, -0.12180651],
       [-0.20371883,  0.09637542,  0.30343946,  0.05377872,  0.03905786],
       [-0.22638348,  0.66394756,  0.05377872,  0.54522132,  0.00267224],
       [ 0.07681744, -0.12180651,  0.03905786,  0.00267224,  0.12980855]])
```

Linear Algebra

```
mat @ inv(mat)
Output:
array([[ 1.00000000e+00,  4.59791778e-16,  1.08466144e-16,
         6.28679353e-17, -1.15104290e-16],
       [ 1.04871403e-16,  1.00000000e+00, -7.39057356e-18,
         4.76991912e-17, -2.78488636e-17],
       [-3.43591815e-17,  8.46236189e-17,  1.00000000e+00,
        -2.53600563e-17, -2.71017556e-17],
       [-9.17010283e-18, -1.01624336e-16,  7.74653173e-18,
         1.00000000e+00,  3.55120708e-17],
       [ 1.32915690e-16, -6.93517700e-17, -1.16881132e-16,
        -2.49601391e-16,  1.00000000e+00]])
```

Linear Algebra

- Commonly used **numpy.linalg** functions



Function	Description
`diag`	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudoinverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system Ax = b for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to `Ax = b`

The End

THE END!!!